Vector ISA Extension for Sparse Matrix-Vector Multiplication

Stamatis Vassiliadis, Sorin Cotofana, and Pyrrhos Stathis

Delft University of Technology, Electrical Engineering Dept.,
P.O. Box 5031, 2600 GA Delft, The Netherlands
{Stamatis,Sorin,Pyrrhos}@Plato.ET.TUDelft.NL

Abstract. In this paper we introduce a vector ISA extension to facilitate sparse matrix manipulation on vector processors (VPs). First we introduce a new Block Based Compressed Storage (BBCS) format for sparse matrix representation and a Block-wise Sparse Matrix-Vector Multiplication approach. Additionally, we propose two vector instructions, Multiple Inner Product and Accumulate ($MIPA$) and LoaD Section (LDS), specially tuned to increase the VP performance when executing sparse matrix-vector multiplications.

1 Introduction

In many areas of scientific computing the manipulation of sparse matrices constitutes the kernel of the solvers. Improving the efficiency of sparse operations such as Sparse Matrix-Vector Multiplication (SMVM) has been and continues to be an important research topic. Several compressed formats for sparse matrix representation [5] and algorithms to improve sparse matrix multiplication performance on parallel [3] and vector machines [4] have been developed. Moreover sparse matrix solving machines, e.g., $(SM)^2$ [1], or Finite Element Method (FEM) [7] computations dedicated parallel architectures, e.g., the White Dwarf [6], SPAR [11], have been proposed.

For reasons related to memory bandwidth and to avoid trivial multiplications with zero values sparse matrices are operated upon on compressed formats. Such compressed matrix formats consist of two main data structures: first, the matrix element values, consisting mainly of the non-zero matrix elements and second, the positional information for the first data structure. A compact representation however implies a loss of data regularity and this deteriorates the performance of vector and parallel processors when operating on sparse formats. For example, as suggested in [11], when executing FEM computations a CRAY Y-MP vector supercomputer operates at less than 33% of its peak floating-point unit throughput.

Up to our best knowledge, with the exception of [8], not much has been done to improve the architectural support that vector processors may provide to sparse matrix multiplication. In this paper we propose a vector ISA extension and an associated sparse matrix organization to alleviate the previously mentioned problem. The main contributions of our proposal can be summarized as follows:

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 708–715, 1999.

- A Block Based Compressed Storage (BBCS) sparse matrix representation format which requires a memory overhead in the order of $\log s$ bits per nonzero matrix element, where s is the vector processor section size, is introduced.
- A Block-wise SMVM scheme to compute the product of an $n \times n$ sparse matrix with a dense vector in $\frac{n}{s}$ vectorizable loops is described. All the trivial with zero multiplications are eliminated and the amount of processed short vectors is substantially reduced.
- Two vector instructions to support the vectorization and execution of the Block-wise SMVM scheme, *Multiple Inner Product and Accumulate* ($MIPA$) and *LoaD Section* (LDS) are proposed.

The presentation is organized as follows: First, in Section 2 we introduce assumptions and preliminaries about sparse matrices and SMVM. Section 3 describes the sparse matrix compact storage format to be used in conjunction with the proposed ISA extension. Section 4 discusses a Block-wise SMVM method and the vector ISA extension. Finally, in Section 5 we draw some conclusions.

2 Problem Statement, Assumptions, & Preliminaries

The definition of the multiplication of a matrix $\boldsymbol{A} = [a_{i,j}]_{i,j=0,1,\ldots,n-1}$ by a vector $\boldsymbol{b} = [b_i]_{i=0,1,\ldots,n-1}$ producing a vector $\boldsymbol{c} = [c_i]_{i=0,1,\ldots,n-1}$ is as follows:

$$\boldsymbol{c} = \boldsymbol{A}\boldsymbol{b}, \qquad c_i = \sum_{k=0}^{n-1} a_{i,k} b_k, \qquad i = 0, 1, \ldots, n-1 \tag{1}$$

Let now consider the execution of the multiplication in Equation (1) on a vector architecture [9]. More in particular we assume a register type of organization, e.g., $IMB/370$ vector facility [2, 10], with the section size of s elements per register. When executed on such a VP the inner loop, i.e, the computation of $c_i = \sum_{k=0}^{n-1} a_{i,k} b_k$, can be vectorized. Ideally, if the section size s is large enough, i.e., $s \geq n$, one loop could be replaced with just one inner product instruction which multiplies the $\boldsymbol{A_i}$ vector, i.e., the i^{th} row of the matrix $\boldsymbol{A}$, with the $\boldsymbol{b}$ vector and produces as result the i^{th} element of the $\boldsymbol{c}$ vector. In practice the section size is usually smaller than n and the $\boldsymbol{A_i}$ and $\boldsymbol{b}$ vectors have to be split into segments of at most s element length to fit in vector registers. Consequently, the computation of c_i will be achieved with a number of vector instructions in the order of $\lceil \frac{n}{s} \rceil$. Although the speedup due to vectorization obviously depends on the section size value, due to issues like lower instruction fetch bandwidth, fast execution of loops, good use of the functional units, VPs perform quite well when $\boldsymbol{A}$ is dense. When operating on sparse matrices however VPs are not as effective because the lack of data regularity in sparse formats leads to performance degradation. Consider for instance that $\boldsymbol{A}$ is sparse and stored in the Compressed Row Storage (CRS)[1]. To process this format each row of non-

[1] The CRS format consists of the following sets: *Value* (all the $a_{i,j} \neq 0$ elements row-wise stored), *Column_Index* (the $a_{i,j} \neq 0$ column positions), and *Row_Pointer* (the

zero values forms a vector and it is accessed by using the index vector provided by the same format. As the probability that the number of non-zero elements in a matrix row is smaller than the VP's section size is rather high many of the vector instructions manipulate short vectors. Consequently, the processor resources are inefficiently used and the pipeline start-up times dominate the execution time. This makes the vectorization poor and constitutes the main reason for VPs performance degradation. An other source for performance degradation might be related to the use of the indexed load/store instructions which, in principle, can not be completed as efficient as a the standard load vector instruction.

The occurrence of short length vectors relates to the fact that vectorization is performed either on row or column direction but not on both. The number of $\boldsymbol{A}$ matrix elements within a vector register can be increased if more than one row (column) is loaded in a vector register at a time. Such vectorization schemes however are not possible assuming the VPs state of the art as such an approach introduces operation irregularity on the elements of the same vector register. Whereas the data irregularity can be easily dealt with by using index based operations and/or masked execution, the operation irregularity requires specialized architectural support and this is exactly what our vector ISA extension does: it enables the operation on multiple matrix rows at once.

3 Sparse Matrix Storage Format

Before describing the proposed vector ISA instruction extension we first introduce a new Block Based Compressed Storage (BBCS) format to be used for the representation of the $\boldsymbol{A}$ sparse matrix.

The $n \times n$ $\boldsymbol{A}$ matrix is partitioned in $\lceil \frac{n}{s} \rceil$ Vertical Blocks (VBs) $\boldsymbol{A^m}$, $m = 0, 1, \ldots, \lceil \frac{n}{s} \rceil - 1$, of at most s columns, where s is the VP section size. For each vertical block $\boldsymbol{A^m}$, all the $a_{i,j} \neq 0$, $sm \leq j < s(m+1)$, $i = 0, 1, \ldots, n-1$, are stored row-wise in increasing row number order. At most one block, the last block, can have less than s columns in the case that n is not a multiple of s. An example of such a partitioning is graphically depicted in Figure 1. In the discussion to follow we will assume, for the simplicity of notations, that n is divisible by s and all the vertical blocks span over s columns.

The rationale behind this partitioning is related to the fact that when computing the $\boldsymbol{Ab}$ product the matrix elements a_{ij} are used only once in the computation whereas the elements of $\boldsymbol{b}$ are used several times depending on the amount of non-zero entries in the $\boldsymbol{A}$ matrix in the corresponding column, as it can be observed in Equation (1). This implies that to increase performance it is advisable to maintain the $\boldsymbol{b}$ values within the execution unit which computes the sparse matrix-vector multiplication for reuse and only stream-in the a_{ij} values. As to each vertical block $\boldsymbol{A^m}$ corresponds an s element section of the $\boldsymbol{b}$-vector, $\boldsymbol{b^m} = [b_{ms}, b_{ms+1}, \ldots, b_{ms+s-1}]$ we can multiply each $\boldsymbol{A^m}$ block with its cor-

pointers in the first two sets to the first non-zero element of each row that contains at least one non-zero element) [5].

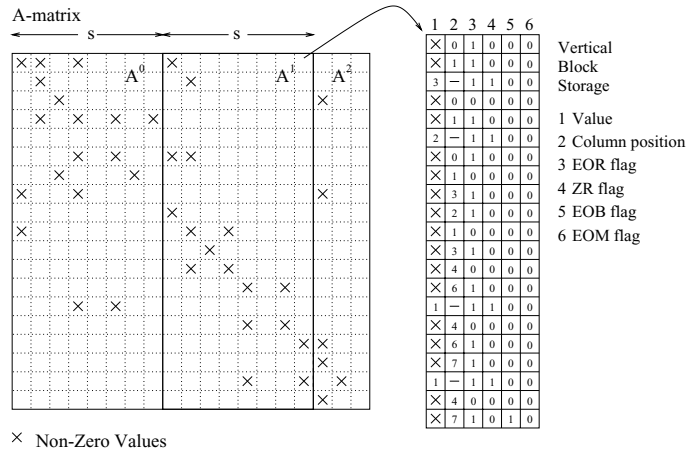

Fig. 1. Block Based Compressed Storage Format

responding $\boldsymbol{b^m}$ section of the $\boldsymbol{b}$-vector without needing to reload any $\boldsymbol{b}$-vector element.

Each $\boldsymbol{A^m}$, $m = 0, 1, \ldots, \frac{n}{s} - 1$, is stored in the main memory as a sequence of 6-tuple entries. The fields of such a data entry are as follows:

1. **Value**: specifies the value of a non-zero $a_{i,j}$ matrix element if $ZR = 0$. Otherwise it denotes the number of subsequent block rows[2] with no non-zero matrix elements.
2. **Column-Position (CP)**: specifies the matrix element column number within the block. Thus for a matrix element $a_{i,j}$ within the vertical block $\boldsymbol{A^m}$ it is computed as $j \mod m$.
3. **End-of-Row Flag (EOR)**: is 1 when the current data entry describes the last non-zero element of the current block row and 0 otherwise.
4. **Zero-Row Flag (ZR)**: is 1 when the current block row contains no non-zero value and 0 otherwise. When this flag is set the $Value$ field denotes the number of subsequent block rows that have no non-zero values.
5. **End-of-Block flag (EOB)**: when 1 it indicates that the current matrix element is the last non-zero one within the VB.
6. **End-of-Matrix flag (EOM)**: is 1 only at the last entry of the last VB of the matrix.

The entire $\boldsymbol{A}$ matrix is stored as a sequence of VBs and there is no need for an explicit numbering of the VBs.

When compared with other sparse matrix representation formats, our proposal requires a lower memory overhead and bandwidth since the index values associated with each $a_{i,j} \neq 0$ are restricted within the VB boundaries and can be

[2] By block row we mean all the elements of a matrix row that fall within the boundaries of the current VB.

represented only with $\log s$ bits instead of $\log n$ bits. The flags can be explicitly 4-bit represented or 3-bit encoded and, depending on the value of s, they may be packed with the CP field on the same byte/word.

The other multiplication operand, the $\boldsymbol{b}$-vector, is assumed to be dense than no special data types or flags are required. The b_k, $k = 0, 1, \ldots, n-1$, values are sequentially stored and their position is implicit. The same applies for the result, i.e., the $\boldsymbol{c}$-vector.

4 Block-Wise SMVM and ISA Extension

Assume a register vector architecture with section size s and the data organization described in Section 3. The $\boldsymbol{Ab}$ product can be computed as $\boldsymbol{c} = \sum_{m=0}^{\frac{n}{s}-1} \boldsymbol{A^m} \times \boldsymbol{b^m}$.

To vectorize each loop computing the $\boldsymbol{A^m} \times \boldsymbol{b^m}$ product and because generally speaking $\boldsymbol{A^m}$ can not fit in one vector register, we have to split each $\boldsymbol{A^m}$ into a number of subsequent VB-sections $\boldsymbol{A_i^m}$ each of them containing at most s elements. Under the assumption that each vertical block $\boldsymbol{A^m}$ is split into $\#s_m$ VB-sections $\boldsymbol{A_i^m}$ the $\boldsymbol{c}$-vector can be expressed as $\boldsymbol{c} = \sum_{m=0}^{\frac{n}{s}-1} \sum_{i=0}^{\#s_m} \boldsymbol{A_i^m} \times \boldsymbol{b^m}$. Consequently, $\boldsymbol{c}$ can be iteratively computed within $\frac{n}{s}$ loops as $\boldsymbol{c_m} = \boldsymbol{c_{m-1}} + \sum_{i=0}^{\#s_m} \boldsymbol{A_i^m} \times \boldsymbol{b^m}$, $m = 0, 1, \ldots, \frac{n}{s} - 1$, where $\boldsymbol{c_m}$ specifies the intermediate value of the result vector $\boldsymbol{c}$ after iteration m is completed and $\boldsymbol{c_{-1}} = \boldsymbol{0}$.

Assuming that $\boldsymbol{A_i^m} = [A_{i,0}^m, A_{i,1}^m, \ldots, A_{i,s-1}^m]$ and $\boldsymbol{b^m} = [b_0^m, b_1^m, \ldots, b_{s-1}^m]$ a standard vector multiplication computes the $\boldsymbol{A_i^m} \times \boldsymbol{b^m}$ inner product as being $c_i^m = \sum_{j=0}^{s-1} A_{i,j}^m b_j^m$ which is the correct result only if $\boldsymbol{A_i^m}$ contains just one row. As one VB-section $\boldsymbol{A_i^m}$ may span over $r_i \geq 1$ rows of $\boldsymbol{A^m}$ the $\boldsymbol{A_i^m} \times \boldsymbol{b^m}$ product should be an r_i element vector. In particular, if $\boldsymbol{A_i^m} = [\boldsymbol{A_i^{m,0}}, \boldsymbol{A_i^{m,1}}, \ldots, \boldsymbol{A_i^{m,r_i-1}}]$, with $\boldsymbol{A_i^{m,j}} = [A_{i,0}^{m,j}, A_{i,1}^{m,j}, \ldots, A_{i,\#r_j-1}^{m,j}]$, $j = 0, 1, \ldots, r_i - 1$, and $\#r_j$ being the number of elements within the row j, $\boldsymbol{c_i^m} = \boldsymbol{A_i^m} \times \boldsymbol{b^m}$ has to be computed as follows:

$$\boldsymbol{c_i^m} = [c_{i,0}^m, c_{i,1}^m, \ldots, c_{i,r_i-1}^m], \qquad c_{i,j}^m = \boldsymbol{A_i^{m,j}} \times \boldsymbol{b^m}, \quad j = 0, 1, \ldots, r_i - 1 \tag{2}$$

Consequently, to compute $\boldsymbol{c_i^m}$, r_i inner products, each of them involving $\#r_j$ elements, have to be evaluated. Moreover when executing the evaluation of $\boldsymbol{A_i^{m,j}} \times \boldsymbol{b^m}$ inner product the "right" elements of $\boldsymbol{b^m}$ have to be selected. Therefore, $c_{i,j}^m$ is evaluated as follows:

$$\begin{aligned} c_{i,j}^m = & A_{i,0}^{m,j} \cdot b_{CP(A_{i,0}^{m,j})}^m + A_{i,1}^{m,j} \cdot b_{CP(A_{i,1}^{m,j})}^m + \ldots \\ & + A_{i,\#r_j-1}^{m,j} \cdot b_{CP(A_{i,\#r_j-1}^{m,j})}^m, \quad j = 0, 1, \ldots, r_i - 1 \end{aligned} \tag{3}$$

As each $c_{i,j}^m$ contributes to the $\boldsymbol{c}$ vector element in the same row position as $\boldsymbol{A_i^{m,j}}$ and row position information is not explicitly memorized in the BBCS format bookkeeping related to index computation has to be performed. Moreover as $\boldsymbol{A_i^m}$ does not contain information about the row position of its first entry, hence not

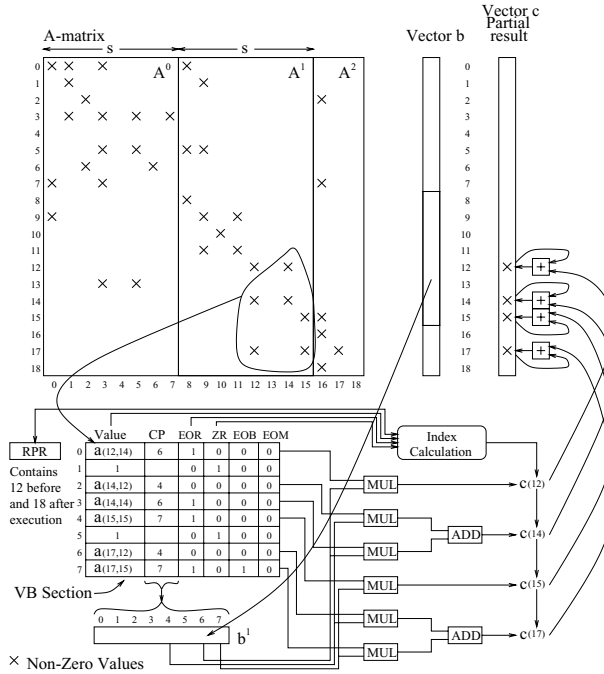

Fig. 2. Sparse Matrix-Vector Multiplication Mechanism

enough information for the calculation of the correct inner product positions is available, a Row Pointer Register (RPR) to memorize the starting row position for $\boldsymbol{A_i^m}$ is needed. The RPR is reset every time the processing of new $\boldsymbol{A^m}$ is initiated and updated by the index computation process.

To clarify the mechanism we present in Figure 2 an example. We assume that the section size is 8, the VB-section contains the last 8 entries of $\boldsymbol{A^1}$, $\boldsymbol{b^1}$ contains the second 8-element section of $\boldsymbol{b}$, and that an intermediate $\boldsymbol{c}$ has been already computed though we depict only the values of $\boldsymbol{c}$ that will be affected by the current step. First the inner product calculation and the index calculation are performed. For the inner product calculation only the $\mathrm{a}(i,j)$ elements with $ZR = 0$ are considered. Initially they are multiplied with the corresponding $\boldsymbol{b^1}$ elements, the CP field is used to select them, and some partial products are obtained. After that, all the partial products within the same row have to be accumulated to form the inner products. As the EOR flags delimit the accumulation boundaries they are used to configure the adders interconnection. The index calculation proceeds with the RPR value and whenever a entry with $ZR = 1$ is encountered RPR is increased with the number in the $Value$ field. When an EOR flag is encountered the RPR is assigned as index to the current inner product and then increased by one. Second the computed indexes are used to select the $\boldsymbol{c}$ elements to whom the computed inner products should be accumulated and the accumulation is performed.

To be able to execute the previously discussed block-wise SMVM we propose the extension of the VP instruction set with 2 new instructions: *Multiple Inner Product and Accumulate* ($MIPA$) and *LoaD Section* (LDS).

$MIPA$ is meant to calculate the inner product $\boldsymbol{A_i^m} \times \boldsymbol{b^m}$. Furthermore, it also performs the accumulation of the $\boldsymbol{c_i^m}$ elements to the $\boldsymbol{c}$-vector values in the corresponding locations. The instruction format is $MIPA\ VR1,VR2,VR3$. The vector register $VR1$ contains $\boldsymbol{A_i^m}$, $VR2$ contains $\boldsymbol{b^m}$, and $VR3$ contains initially those elements of the $\boldsymbol{c}$-vector that correspond to the non-empty rows of $\boldsymbol{A_i^m}$ and after the instruction execution is completed the updated values of them.

LDS is meant to load an $\boldsymbol{A_i^m}$ VB-section from the main memory. The instruction format is $LDS\ @A,VR1,VR2$. $@A$ is the address of the first element of the VB-section to be loaded. After an LDS instruction is completed $VR1$ contains all the non-zero elements of the VB-section starting at $@A$, $VR2$ contains the indexes of the VB-section rows with non-zero elements (to be used later on as an index vector to load and store a $\boldsymbol{c}$ section), and the Column Position Register (CPR), a special vector register, is updated with the CP and flag fields of the corresponding elements in $VR1$. To execute the LDS instruction the VP Load Unit has to include a mechanism to analyze the BBCS flags in order to compute the indexes and filter the entries with $ZR = 1$. In this way $VR1$ always contains only nonzero matrix elements and no trivial calculations will be performed.

Even though the previously discussed approach guarantees an efficient filling of the vector registers, it may suffer a performance degradation due to the use of indexed vector load/store instructions as such operations, depending on the implementation of the main memory and/or load unit, may create extra overhead. However, the use of indexed load/stores can be avoided if instead of partitioning the VBs in s-element VB-sections, they are divided in $\lceil \frac{n}{ks} \rceil$ ks-element hight *segments*[3] where $k \geq 1$. To support this new division the BBCS has to include an extra flag, the End of Segment (EOS) flag. This flag is 1 for the entries which describe the last element of a segment. Under this new assumption when a VB-section is loaded with the LDS instruction the loading stops after s nonzero elements or when encountering the EOS, EOB, or EOM flag. By restricting the VB-section's row span within a VB segment we guarantee that all the $\boldsymbol{A_i^m} \times \boldsymbol{b^m}$s will contribute only to elements within a specific ks-element wide section of the vector $\boldsymbol{c}$. Consequently, the $\boldsymbol{c}$-vector can be manipulated with standard load/store instructions.

5 Conclusions

In this paper we proposed a vector ISA extension and an associated sparse matrix organization to alleviate some of the problems related to sparse matrix computation on vector processors, e.g., inefficient functional unit utilization due to short vector occurrence and increased memory overhead and bandwidth.

[3] All of them are $s \times ks$ segments except the ones on the right and/or bottom edge of the matrix which might be truncated if the dimension of the matrix n is not divisible by s.

First we introduced a Block Based Compressed Storage (BBCS) sparse matrix representation format which requires a memory overhead in the order of $\log s$ bits per nonzero matrix element, where s is the vector processor section size. Additionally, we described a Block-wise SMVM scheme to compute the product of an $n \times n$ sparse matrix with a dense vector in $\frac{n}{s}$ vectorizable loops. To support the vectorization of the Block-wise SMVM scheme two new vector instructions, *Multiple Inner Product and Accumulate* ($MIPA$) and *LoaD Section* (LDS) were proposed. They eliminate all the trivial multiplications with zero and substantially reduce the amount of processed short vectors. Implementation and quantitative evaluations of the performance of the proposed schemes via simulations on sparse matrices benchmarks constitutes the subject of future research.

References

[1] H. Amano, T. Boku, T. Kudoh, and H. Aiso. $(SM)^2$-II: A new version of the sparse matrix solving machine. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 100–107, Boston, Massachusetts, June 17–19, 1985. IEEE Computer Society TCA and ACM SIGARCH.

[2] W. Buchholz. The IBM System/370 vector architecture. *IBM Systems Journal*, 25(1):51–62, 1986.

[3] R. Doallo, J. T. no, and F. Hermo. Sparse matrix operations in vector and parallel processorse. *High Performance Computing*, 3:43–52, 1997.

[4] I. S. Duff. The use of vector and parallel computers in the solution of large sparse linear equations. In *Large scale scientific computing. Progress in Scientific Computing Volume 7*, pages 331–348, Boston, MA, USA, 1986. Birkhäuser.

[5] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, UK, 1986.

[6] A. W. et al. The white dwarf: A high-performance application-specific processor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 212–222, Honolulu, Hawaii, May–June 1988. IEEE Computer Society Press.

[7] T. J. R. Hughes. *The Finite Element Method*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[8] R. N. Ibbett, T. M. Hopkins, and K. I. M. McKinnon. Architectural mechanisms to support sparse vector processing. In *Proceedings of the 16th ASCI*, pages 64–71, Jerusalem, Israel, June 1989. IEEE Computer Society Press.

[9] P. M. Kogge. *The Architecture of Pipelined Computers*. McGraw-Hill, New York, 1981.

[10] A. Padegs, B. B. Moore, R. M. Smith, and W. Buchholz. The IBM System/370 vector architecture: Design considerations. *IEEE Transactions on Computers*, 37:509–520, 1988.

[11] V. E. Taylor, A. Ranade, and D. G. Messerschitt. SPAR: A New Architecture for Large Finite Element Computations. *IEEE Transactions on Computers*, 44(4):531–545, April 1995.